Wartbare Silberkugeln

Single-Page-Anwendungen mit AngularJS und Java EE

Torsten Fink, Till Hermsen, Philipp Kumar

Die Java EE ist mit dem serverzentrierten JSF-Framework-Standard für grafische Benutzungsoberflächen nur schwach ausgerüstet für moderne Single-Page-Webanwendungen. Moderne JavaScript-Bibliotheken ermöglichen einen deutlich höheren Benutzerkomfort. AngularJS ist ein Framework, das mit seinem sauberen deklarativen MVC-Konzept und seiner modularen Komponentenarchitektur die Erstellung und Wartung komplexerer Oberflächen ermöglicht. Über REST ist auch der Anschluss eines Java EE-Backends problemfrei möglich.

JavaScript-basierte GUI-Bibliotheken

Anwender von Webanwendungen stellen heutzutage große Anforderungen an Interaktivität und Komfort. Wird bei jedem Klick ein durchgängiger Roundtrip über den Server durchgeführt, fühlt sich die Anwendung behäbig an. Damit wird JSF als Framework für Webanwendungen immer ungeeigneter, insbesondere, wenn die Anwendung im Internet gegen Marktbegleiter bestehen muss. JavaScript-basierte Frameworks versprechen da Besserung. Allerdings haben viele Java-Entwickler immer noch Vorbehalte gegen die Erstellung komplexer Oberflächen mit JavaScript. Ursache hierfür sind Spracheigenschaften wie der globale Namensraum und das dynamische Laufzeitverhalten, welche die Fehleranfälligkeit erhöhen.

Wir sind allerdings der Meinung, dass sich die JavaScript-basierten Frameworks inzwischen so stark weiter entwickelt haben, dass sie sich auch für große Projekte eignen. Dies gilt insbesondere für das Open-Source-Framework AngularJS, das von der Firma Google entwickelt wird. Auch die Kombination mit einem Java EE-basierten Backend stellt kein Problem mehr dar.

In einem Schnelldurchlauf stellen wir anhand praktischer Beispiele die Konzepte von AngularJS vor. Die Beispiele in diesem Artikel sind größtenteils unserem neuen Buch [Kum14] entnommen, das sich auch mit weiterführenden Themen beschäftigt, wie dem geeigneten Aufsetzen eines Build-Systems. Der Code ist auf Github zu finden [Exam].

Initialisierung einer AngularJS-Anwendung

Listing 1 zeigt eine typische Initialisierung einer Single-Page-Anwendung mit AngularJS. Unten im Rumpf (`body`) werden die benötigten JavaScript-Bibliotheken geladen. Hierzu gehört AngularJS selbst und der anwendungsspezifische Code, der in `app.js` liegt. Nachdem die Seite geladen wurde, sucht AngularJS in dem HTML-Tag nach dem Attribut `ng-app`. In dem Beispiel findet es den Wert `dailyPlanner`. Nun wird nach einem gleichnamigen Modul gesucht und dieses aktiviert. Die Modulimplementierung findet sich in `app.js`. Das Modul bietet alle Elemente an, die die Anwendung benötigt.

```
Einstiegsseite:
<html ng-app="dailyPlanner">
  <head>
    ...
  </head>
  <body>
    ...
    <script type="text/javascript" src="/angular.js"></script>
    <script type="text/javascript" src="/app.js"></script>
  </body>
</html>

app.js:
angular.module("dailyPlanner"])
  .controller(...);
```

Listing 1:Initialisierung einer AngularJS-Anwendung

MVC im Browser

Wir möchten nun eine Liste von Aufgaben darstellen, Abbildung 1 zeigt die Oberfläche. AngularJS bietet eine Model-View-Controller-Architektur, wobei die View größtenteils direkt in der HTML-Oberfläche realisiert wird, die Model- und Controller-Elemente in JavaScript.

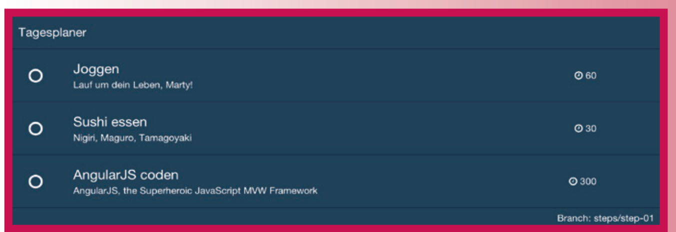

Abb. 1: Liste von Aufgaben

Listing 2 zeigt die View der Taskliste, wobei alle Styling-Elemente entfernt wurden. Über das Attribut `ng-controller` wird ein Controller einem Abschnitt in der HTML-Struktur zugeordnet, hier konkret der `taskListController`. Der Controller spannt einen Kontext auf, in dem Daten und Operationen verfügbar sind, die in der View eingesetzt werden können. Hier enthält der Kontext ein Modell `tasks`, das aus einer Liste von Task-Objekten besteht. Die Anweisung `ng-repeat` iteriert über diese Liste und gibt dabei alle Daten als **<li>**-Elemente aus. Solche Anweisungen werden als Direktiven bezeichnet. Mit dem Konstrukt `{{task.title}}` werden Modelldaten direkt in HTML übernommen.

```
<div ng-controller="taskListController">
  <ul>
    <li ng-repeat="task in tasks">
      {{ task.title }} <br>
      {{ task.description }} </p>
    </li>
  </ul>
</div>
```

Listing 2: View einer Taskliste

```
.controller("taskListController",
    function ($scope) {
  $scope.tasks = [
    {
      title: "Joggen",
      description: "Lauf um dein Leben, Marty!",
      duration: 60,
```

```
        done: false
      },
      .... // hier sind die restlichen Tasks
    ];
  })
```

Listing 3: Model und Controller der Taskliste

Listing 3 zeigt den JavaScript-Code zum Aufsetzen des Controllers, den wir vorher ausgelassen hatten. Der Controller wird unter dem Namen `taskListController` angemeldet. Wird ein Exemplar des Controllers benötigt, wird die assoziierte Funktion ausgeführt. Diese erhält als Parameter ein `$scope`-Objekt, das den zugeordneten Kontext darstellt. Diesem Kontext werden Daten und Operationen hinzugefügt, die von der View benutzt werden können. In unserem Beispiel wird zur Verdeutlichung des Konzepts unter dem Namen `tasks` eine statische Liste von Aufgaben registriert.

Damit haben wir die HTML-Liste aller Aufgaben an das JavaScript-Modell `tasks` gebunden. Wird durch eine Operation das Objekt `tasks` geändert, passt sich ohne weiteres Zutun des Entwicklers die HTML-Liste an. Bindet man Modelldaten an HTML-Elemente, die Eingaben erlauben, dann werden bei Nutzereingaben automatisch auch die Modelldaten geändert. Die Bindung ist also bidirektional.

Formulare

Abbildung 2 zeigt ein Formular zur Eingabe der Attribute einer Aufgabe. Das Feld für den Titel ist noch leer. Da es obligatorisch ist, wird es rot dargestellt und ein Informationstext angezeigt.

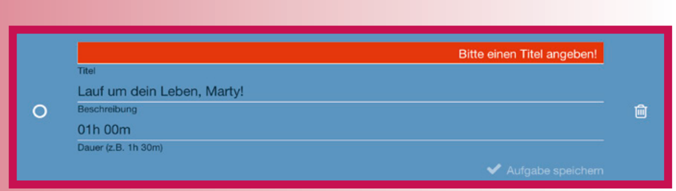

Abb. 2: Bearbeiten einer Aufgabe

Listing 4 zeigt einen Ausschnitt der zugehörigen View. Das `<form>`-Element ist nicht das Standard-HTML-Element, sondern wird von AngularJS durch eine eigene Direktive ausgetauscht, die neben der Validierung unter anderem auch dafür sorgt, dass ein Submit nicht über einen Browser-Reload den Zustand der Seite zurücksetzt.

```
<form name="editTaskForm" ng-submit="saveTask()">
  <input id="task-title-input" type="text" name="title"
         ng-model="selectedTask.title"
         ng-required="true"
         ng-minlength="3" >
  <label for="task-title-input"> Titel
     <span ng-show="editTaskForm.title.$error.required">
       Bitte einen Titel angeben.
     </span>
     <span ng-show="editTaskForm.title.$error.minlength">
       Bitte mindestens 3 Zeichen angeben.
     </span>
  </label>
  ...
  <button type="submit" ng-disabled="editTaskForm.$invalid">
    Aufgabe speichern
  </button>
</form>
```

Listing 4: Formular zum Bearbeiten einer Aufgabe

Das `<form>`-Element wird von einem eigenen Controller verwaltet. Mit dem Attribut `name` wird er in unserem Beispiel mit `editTaskForm` benannt. Über `ng-submit` wird bei einem Submit eine JavaScript-Funktion ausgeführt. Diese muss in dem Kontext des Formulars vorhanden sein. Hier ist dies die Funktion `saveTask()`, die sich alle Daten aus dem Datenmodell zusammen sammelt und asynchron an den Server schickt. Wie dies funktioniert, zeigen wir später.

Das Textfeld ist über das Attribut `ng-model` mit dem Datenmodellelement `selectedTask.title` gebunden. Über `ng-required` ist spezifiziert, dass eine Eingabe notwendig ist. In dem zugehörigen Label wird eine Warnnachricht angezeigt, solange das Textfeld noch nicht gesetzt ist. Hierfür wird das Attribut `ng-show` benutzt, welches ein Element nur dann anzeigt, wenn der gesetzte Ausdruck zu `true` ausgewertet wird. Als Ausdruck wird auf den Form-Controller zugegriffen, der den Validierungszustand von jedem Feld verwaltet. Zusätzlich soll der Titel mindestens drei Zeichen lang sein. Hierfür wird das Attribut `ng-minlength` eingesetzt. Die Änderungen können erst dann gespeichert werden, wenn alle Felder gültig gesetzt sind. Der Speichern-Knopf wird mit `ng-disabled` solange deaktiviert, bis das Formular gültig ist.

AngularJS bietet noch weitere Mechanismen zur Validierung an, insbesondere auch die Unterstützung regulärer Ausdrücke. Die MVC-Architektur ermöglicht es zusätzlich, eigene Validierungslogik einfach zu integrieren.

Filter

Die vorgestellte bidirektionale Bindung funktioniert gut bei Daten wie Name oder Straße, also bei einer direkten Zuordnung einer HTML-Komponente zu einem Attribut. Manchmal ist für die Bindung aber ein Übersetzungsschritt notwendig.

Als Beispiel schauen wir uns die Dauer einer Aufgabe an. Der Benutzer kann diese als `2h 30m` spezifizieren. Im Datenmodell wird aber die Anzahl der Minuten als Zahl gespeichert, also als 150. AngularJS hilft hier mit sogenannten *Filtern* bei der Darstellung solcher Daten. Listing 5 zeigt ihre Anwendung. Im View wird in HTML per `|` der einzusetzende Filter per Namen spezifiziert. Der zugehörige Code wird in einem Modul per `filter()` angemeldet. Bei der Umkehrrichtung, also zum Beispiel von einem Textfeld zu dem Datenmodell, bieten Filter allerdings keine Unterstützung.

```
In HTML:

{{ task.duration | formatDuration }}

In JavaScript:

  .filter("formatDuration", [
    function () {
      return function (input) {
        var output, hours, minutes;
        ... Berechnung der Darstellung
        return outputString;
      };
    }
  ]);
```

Listing 5: Übersetzung zwischen Model und View mit Filtern

Templating

Webseiten bestehen häufig aus statischen und dynamischen Anteilen. Über einen Templating-Mechanismus ermöglicht

AngularJS, diese voneinander zu trennen. In Listing 6 ist eine Datei mit den statischen Anteilen zu sehen. Über die Direktive **ng-include** wird der dynamische Anteil nachgeladen. Aktuelle Suchmaschinen haben kein Problem, auch solche dynamisch erzeugten Seiten zu indizieren.

```
<!DOCTYPE html>
<html ng-app="dailyPlanner">
  ...
<body>
  <div class="container" ng-include="'partials/main.html'"></div>
  ...
</body>
```

Listing 6: Trennung statischer und dynamischer Anteile

Navigation in Single-Page-Anwendungen

Die nächste Stufe nach einem einfachen Templating ist eine komplexere Navigation, bei der der Inhalt des Templates dynamisch ausgetauscht wird, zum Beispiel durch eine Navigation über ein Menü. AngularJS bietet eine Erweiterung namens **ng-route**, mit der per URLs der Inhalt eines Templates ausgewechselt werden kann. Listing 7 zeigt das Konzept. Im oberen Teil ist ein exemplarisches Menü zu sehen, in dem über **<a>**-Elemente Referenzen auf andere Bereiche angegeben sind. Im unteren Teil wird konfiguriert, welche HTML-Fragemente in das Template abhängig von der URL geladen werden. Auf diese Weise lassen sich komplexere Anwendungen über unterschiedliche Seiten strukturieren, ohne die hohe Interaktivität von Single-Page-Anwendungen zu verlieren.

```
In HTML:
<ul>
  <li><a href="#/main">Hauptseite</a></li>
  <li><a href="#/admin">Administration</a></li>
</ul>

In JavaScript:
$routeProvider.
  when("/main",  {templateUrl: "partials/main.html"}).
  when("/admin", {templateUrl: "partials/admin.html"}).
  otherwise({redirectTo: "/main"});
```

Listing 7: Navigation über URLs

Modularisierung der GUI

Anwendungen werden schnell sehr komplex, insbesondere Geschäftsanwendungen. Das Modulkonzept von AngularJS ermöglicht, die gesamte Oberflächenlogik in klare Subkomponenten zu zergliedern. Ein Modul definiert sich durch einen Namen, eine Liste benötigter Module und eine Menge von Elementen.

Mit Dependency-Injection wird sichergestellt, dass der Code die spezifizierten Abhängigkeiten nicht verletzen kann. So wird in Listing 3 der Parameter **$scope** an die Funktion übergeben. Die Kombination von hierarchischen Kontexten und Dependency-Injection löst die Probleme, die durch den globalen Namensraums in JavaScript entstehen.

AngularJS bietet etliche Modulelemente an. Die wichtigsten sind die folgenden:

- *Dienste* sind Funktionen, die über ein Factory-Pattern konfigurierbar erzeugt werden,
- *Direktiven* kapseln komplexere Oberflächenelemente, die in Views wiederverendet werden können,
- *konstante* und *variable Werte* und
- *Controller* und *Filter*, die schon vorgestellt wurden.

Automatisierte Tests

AngularJS unterstützt die Erstellung automatisiert ausführbarer Tests, und zwar sowohl auf Modul- als auch auf Systemebene. Listing 8 zeigt einen Modultest, der mit dem BDD-Framework Jasmine entwickelt wurde. Zu erkennen ist, wie vor jedem Test mittels **beforeEach** das Modul **filters** geladen wird. Danach stehen alle Elemente des Moduls per Dependency-Injection für Tests zur Verfügung. In dem Beispiel testen wir den oben vorgestellten Filter **formatDuration** zur Anzeige der Dauer einer Aufgabe.

Neben Modultests ist auch das Testen der gesamten Anwendung über ihre externen Schnittstellen wichtig. AngularJS bietet hierfür ein Framework, das ein Fernsteuern eines virtuellen Browsers und Zugriff auf AngularJS-Komponenten ermöglicht. Listing 9 zeigt ein Beispiel zur Verdeutlichung des Konzepts. Zunächst wird auf die Seite navigiert, die alle aktuellen Aufgaben des Benutzers auflistet. Über den Zugriff auf das interne Datenmodell **taskList** wird die aktuelle Anzahl an Aufgaben ermittelt. Dann wird über die Oberfläche eine neue Aufgabe hinzugefügt und danach geprüft, ob an der Oberfläche eine zusätzliche Aufgabe angezeigt wird.

```
describe("filters", function () {
  beforeEach(module("filters"));

  describe("formatDuration filter", function () {
    it("should format 150 minutes to '02h 30m'",
        inject(function ($filter) {
      var input, formattedInput;
      input = 150;
      formattedInput = $filter("formatDuration")(input);
      expect(formattedInput).toEqual("02h 30m");
    }));
  });
});
```

Listing 8: Modultests mit Jasmine

```
it('should add a new task to the task-list', function () {
  browser().navigateTo("/dailyplanner/pages/index.html");

  element("#task-list", "add new task").query(
      function (taskList, done) {
    // count existing tasks
    var numberOfTasks = taskList.children().length;

    // add new task
    input("newTaskTitle").enter("new task");
    element("#add-new-task-form button:submit", "add new task").click();

    browser().reload();

    expect(repeater("#task-list li", "").count()).toBe(
      numberOfTasks + 1);
    done();
  });
});
```

Listing 9: Ende-zu-Ende-Tests

Was ist mit GUI-Komponenten?

AngularJS enthält keine Oberflächenkomponenten zusätzlich zu dem HTML-Standard, wie Baumstrukturen, Menüs, interaktive Karten und so weiter. Hierfür können vorhandene Erweiterun-

gen, wie das Bootstrap-Framework, eingesetzt werden. Da diese aber selber JavaScript-Bibliotheken und meistens auch eigene Konzepte mitbringen, besteht die Gefahr von Architekturbrüchen. Deshalb gibt es etliche Bibliotheken, die zusätzliche Komponenten anbieten, welche an AngularJS angepasst sind. Einen guten Überblick gibt [AngularUI].

Alles nichts ohne Server

Auch die schönste Weboberfläche benötigt einen Servergegenpart. AngularJS setzt hier auf eine REST-Architektur mit JSON als Datenformat. Mit der Erweiterung **`ng-resource`** lassen sich einfach Server anschließen. Listing 10 zeigt dies exemplarisch für die Verwaltung der Aufgaben eines Nutzers. Über die URL **`http://server.com/app/rest/plan`** stellt die Anwendung die Verwaltung aller Aufgaben des angemeldeten Benutzers bereit. Der letzte Teil der URL, **`plan`**, bezeichnet die Ressource, während das Präfix für alle Ressourcen der Anwendung gleich und in der Konstanten **`basePath`** abgelegt ist. Der Service **`planResource`** soll den Zugriff realisieren. Es wird eine Fabrikfunktion registriert, die mit der Erweiterung **`$resource`** ein JavaScript-Objekt erzeugt, welches den Service bereitstellt.

Die Controller-Logik bekommt nun dieses Service-Objekt herein gereicht, wie in Listing 10 zu sehen ist. Zunächst lädt die Controller-Initialisierung über einen **`query()`**-Aufruf vom Server alle aktuellen Aufgaben. Auch wenn der Code sequenziell erscheint, werden die Fernzugriffe asynchron im Hintergrund durchgeführt. Zurückgeliefert wird ein Promise-Objekt, das nach Eintreffen der Daten vom Server befüllt wird. Danach ist der Code für das Löschen einer Aufgabe zu sehen. Hierfür wird zunächst das interne Datenmodell angepasst und dieses dann per **`save()`**-Aufruf asynchron an den Server geschickt. Über Rückrufmethoden kann der Entwickler auf den Abschluss der Operation reagieren.

Listing 11 zeigt das Gegenstück auf dem Server. Die Klasse **`DailyPlannerRest`** ist eine zustandslose EJB, die per JAX-RS-Annotationen als REST-Schnittstelle fungiert. Die Methode **`getDailyPlan()`** realisiert die GET-Zugriffe für das Laden der Aufgaben, während **`saveDailyPlan()`** durch ein POST angestoßen wird und Änderungen in die Datenbank schreibt. Die eigentlichen Datenbankoperationen werden in einer anderen Architekturschicht von der EJB **`DailyPlanDao`** implementiert.

```
// Registrierung des Serverzugriff als Service an einem Modul
.factory("planResource", function ($resource, basePath) {
    return $resource(basePath+"plan");
  }
]);

// Einsatz im Controller
.controller("taskListController",
    function ($scope, planResource, $filter, $log) {
  ...
  $scope.tasks = planResource.query();
  ...
  $scope.deleteTask = function (taskIndex) {
    $scope.tasks.splice(taskIndex, 1);
    planResource.save({}, $scope.tasks, function () {
      // success
      $log.info("Saved my daily plan.");
    }, function () {
      // failure
      $log.error("Could not save my daily plan.");
    });
  }
  ...
};
```

Listing 10: Zugriff auf den Server

```
@Path("/")
@Stateless
public class DailyPlannerRest {
  @EJB
  private DailyPlanDao dailyPlanDao;

  @GET
  @Path("/plan")
  @Produces({"application/json"})
  public TaskDto[] getDailyPlan() {
    return dailyPlanDao.findTasksOfDailyPlanForUser(getUserId());
  }
  @POST
  @Path("/plan")
  @Consumes({"application/json"})
  public void saveDailyPlan(TaskDto[] taskDtos) {
    dailyPlanDao.saveDailyPlanForUser(getUserId(), taskDtos);
  }
  private String getUserId() { ... }
}
```

Listing 11: REST mit der Java EE

Fazit

AngularJS ist ein Framework, das die Entwicklung auch komplexer Oberflächen in JavaScript ermöglicht. Es ist inzwischen ausgereift und wird von Google aktiv weiterentwickelt. Der Einsatz der Java EE auf der Serverseite gestaltet sich aufgrund des REST-Paradigmas mit JAX-RS problemfrei.

Literatur und Links

[AngularUI] **`http://angular-ui.github.io`**

[Exam] Beispieldateien auf Git-Hub, dailyplanner-angularjs, **`https://github.com/akquinet/dailyplanner-angularjs`**

[Kum14] Ph. Kumar u. a., Rich Web Apps mit AngularJS und Java EE, iTunes-Buch, Akquinet, 2014, **`https://itunes.apple.com/de/book/rich-web-apps-mit-angularjs/id847457516`**

Dr. Torsten Fink ist Geschäftsführer der Berliner Einheit der akquinet. Neben der Leitung von Java EE-Projekten führt er Architektur- und Technologieberatungen durch.
E-Mail: torsten.fink@akquinet.de

Till Hermsen arbeitet als Entwickler bei der akquinet. Er beschäftigt sich schwerpunktmäßig mit mobilen Webanwendungen.
E-Mail: till.hermsen@akquinet.de

Philipp Kumar ist Leiter des Competence Center für mobile Lösungen bei der akquinet. Seine Schwerpunkte liegen in der Anforderungsanalyse mobiler Systeme, auf Android-basierten Anwendungen und auf der Integration mobiler Lösungen in Unternehmensinfrastrukturen.
E-Mail: philipp.kumar@akquinet.de